Attorney Docket No.: 20949P-001500US
Client Reference No.: 22

PATENT APPLICATION

APPARATUS TO OFFLOAD AND ACCELERATE PICO CODE PROCESSING RUNNING IN A STORAGE PROCESSOR

Inventors: Ryan Taylor Herbst, a citizen of the United States, residing at
1355 Sierra Street
Redwood City, CA 94061

James L. Cihla, a citizen of the United States, residing at
7075 Brooktree Way
San Jose, CA 95120

Rahim Ibrahim, a citizen of Malaysia, residing at
467 Carmelita Drive
Mountain View, CA 94040

James L. Vuong, a citizen of the United States, residing at
412 Cirrus Avenue
Sunnyvale, CA 94087

Assignee: Candera Inc.
673 South Milpitas Blvd.
Milpitas, CA 95035

Entity: Small business concern

TOWNSEND and TOWNSEND and CREW LLP
Two Embarcadero Center, 8th Floor
San Francisco, California 94111-3834
Tel: 650-326-2400

APPARATUS TO OFFLOAD AND ACCELERATE PICO CODE PROCESSING RUNNING IN A STORAGE PROCESSOR

CROSS-REFERENCES TO RELATED APPLICATIONS

[0001] The present application claims priority to U.S. Provisional Application No. 60/422,109 titled "Apparatus and Method for Enhancing Storage Processing in a Network-Based Storage Virtualization System" and filed October 28, 2002, which is incorporated herein by reference.

STATEMENT AS TO RIGHTS TO INVENTIONS MADE UNDER FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

[0002] NOT APPLICABLE

REFERENCE TO A "SEQUENCE LISTING," A TABLE, OR A COMPUTER PROGRAM LISTING APPENDIX SUBMITTED ON A COMPACT DISK.

[0003] NOT APPLICABLE

BACKGROUND OF THE INVENTION

[0004] The present invention relates to storage area networks (SANs). In particular, the present invention relates to an offload processor in a storage server.

[0005] FIG. 1 is a block diagram of a storage area network (SAN) system 10. The SAN system 10 includes a host 12, a network 14, a storage server 16, and a storage system 18. The host 12 generally includes a computer that may be further connected to other computers via the network 14 or via other connections. The network 14 may be any type of computer network, such as a TCP/IP network, an Ethernet network, a token ring network, an asynchronous transfer mode (ATM) network, a Fibre Channel network, etc. The storage system 18 may be any type of storage system, such as a disk, disk array, RAID (redundant array of inexpensive disks) system, etc.

[0006] The storage server 16 generally transparently connects the host 12 to the storage system 18. More specifically, the host 12 need only be aware of the storage server 16, and the storage server 16 takes responsibility for interfacing the host 12 with the storage system 18. Thus, the host 12 need not be aware of the specific configuration of the storage system 18. Such an arrangement allows many of the storage management and configuration functions to be offloaded from the host.

[0007] Such offloading allows economies of scale in storage management. For example, when the storage system 10 has multiple hosts on the network 14 and the components of the storage system 18 are changed, all the hosts need not be informed of the change. The change may be provided only to the storage server 16.

[0008] Similar concepts may be applied to other storage system architectures and arrangements such as networked attached storage (NAS), etc.

[0009] It is advantageous for the storage server 16 to quickly perform its SAN network processing functions. Such functions include semaphore management, out-of-order frame processing, and timer management.

[0010] Semaphore management involves managing semaphores that control access to data space. For example, if one process thread is accessing a particular data space of the storage system 18, then no other process threads should access that data space until the first process thread has completed its access. Otherwise the second process thread could alter the data in the data space in a detrimental manner. Semaphore management is typically performed in software.

[0011] Out-of-order frame processing involves re-arranging frames received out of order by the storage server 16. For example, if the storage server 16 is accessing data that is stored on more than one disk 18, the data from one disk 18 may arrive before the data from another disk 18. The host 12 expects the data in a particular order, so the storage server 16 typically re-orders the data before it is forwarded on to the host 12. Frame re-ordering is typically performed in software.

[0012] Timer management involves creating, checking, and stopping timers related to various activities that the storage server 16 performs. For example, the storage server 16 sets a timer when accessing an element of the storage system 18. If the element fails to respond, the timer

expires, triggering action by the storage server 16 to re-access the element, to perform diagnostic processing, or to otherwise respond to the failure. Timer management is typically performed in software.

[0013] Typical implementations of the above three functions may be inefficient. The software implementing each function typically occupies the general code space in the storage server 16. Such code space is a limited resource that it is often advantageous to conserve. The execution of such software requires processor overhead.

[0014] Aspects of the present invention are directed toward improving the operation of these three functions in the storage server 16.

BRIEF SUMMARY OF THE INVENTION

[0015] Aspects of the present invention are directed toward a co-processor that offloads selected functions of a network processor operating in a storage environment.

[0016] According to one aspect of the present invention, the co-processor includes semaphore circuitry that receives a signal from the network processor and controls a semaphore related to the signal for locking and unlocking access to data. The semaphore circuitry manages a queue of access requests for a particular semaphore, enabling ordered access to the semaphore.

[0017] According to another aspect of the present invention, the co-processor includes ordering circuitry that tracks an order of incoming frames received by the network processor and controls an order of outgoing frames transmitted by the network processor. When the network processor receives an incoming frame, it checks with the co-processor to see if the incoming frame is in order. If so, it transmits the frame immediately without having to perform involved memory accesses. If not, it stores the frame for future transmission.

[0018] According to still another aspect of the present invention, the co-processor includes timer circuitry that manages timers as requested by the network processor and that generates a timing result when one of the timers is completed. The timers are arranged in a doubly-linked list, which reduces overhead and enables efficient insertion of a new timer into the doubly-linked list. Various granularities of timers are provided.

[0019] In this manner, the co-processor improves the performance of the network processor by offloading these selected functions.

[0020] A more complete understanding of the present invention may be gained from the following figures and related detailed description.

BRIEF DESCRIPTION OF THE DRAWINGS

[0021] FIG. 1 is a block diagram of a storage area network system.

[0022] FIG. 2 is a block diagram of a storage server according to an embodiment of the present invention.

[0023] FIG. 3 is a block diagram of a semaphore manager according to an embodiment of the present invention.

[0024] FIG. 4 is a diagram of a hash structure used by the semaphore manager of FIG. 3.

[0025] FIG. 5 is a diagram of a semaphore structure used by the semaphore manager of FIG. 3.

[0026] FIG. 6 is a flow diagram for the semaphore request command executed by the semaphore manager of FIG. 3.

[0027] FIG. 7 is a flow diagram for the semaphore release command executed by the semaphore manager of FIG. 3.

[0028] FIG. 8 is a flow diagram for the thread exit command executed by the semaphore manager of FIG. 3.

[0029] FIG. 9 is a block diagram of an ordering processor according to an embodiment of the present invention.

[0030] FIG. 10 is a diagram of a queue head structure used by the ordering processor of FIG. 9.

[0031] FIG. 11 is a diagram of a frame structure used by the ordering processor of FIG. 9.

[0032] FIG. 12 is a diagram of a queue initialization command used by the ordering processor of FIG. 9.

[0033] FIG. 13 is a diagram of a frame pop command used by the ordering processor of FIG. 9.

[0034] FIG. 14 is a diagram of a frame received command used by the ordering processor of FIG. 9.

[0035] FIG. 15 is a diagram of a frame poll command used by the ordering processor of FIG. 9.

[0036] FIG. 16 is a diagram of a frame transmit command used by the ordering processor of FIG. 9.

[0037] FIG. 17 is a flow diagram for the queue initialization command executed by the ordering processor of FIG. 9.

[0038] FIG. 18 is a flow diagram for the frame pop command executed by the ordering processor of FIG. 9.

[0039] FIG. 19 is a flow diagram for the first part of the frame received command executed by the ordering processor of FIG. 9.

[0040] FIG. 20 is a flow diagram for the second part of the frame received command executed by the ordering processor of FIG. 9.

[0041] FIG. 21 is a flow diagram for the frame poll command executed by the ordering processor of FIG. 9.

[0042] FIG. 22 is a flow diagram for the frame transmit command executed by the ordering processor of FIG. 9.

[0043] FIG. 23 is a block diagram of a timer manager according to an embodiment of the present invention.

[0044] FIG. 24 is a diagram of a restart timer command used by the timer manager of FIG. 23..

[0045] FIG. 25 is a diagram of a doubly-linked list used by the timer manager of FIG. 23.

[0046] FIG. 26 is a diagram of a start timer command used by the timer manager of FIG. 23.

[0047] FIG. 27 is a diagram of a stop timer command used by the timer manager of FIG. 23.

[0048] FIG. 28 is a diagram of a restart timer command used by the timer manager of FIG. 23.

[0049] FIG. 29 is a diagram of a read expired command used by the timer manager of FIG. 23.

[0050] FIG. 30 is a diagram of a clear expired command used by the timer manager of FIG. 23.

[0051] FIG. 31 is a flow diagram of the start timer command executed by the timer manager of FIG. 23.

[0052] FIG. 32 is a flow diagram of the stop timer command executed by the timer manager of FIG. 23.

[0053] FIG. 33 is a flow diagram of the restart timer command executed by the timer manager of FIG. 23.

[0054] FIG. 34 is a flow diagram of the clear expired command executed by the timer manager of FIG. 23.

[0055] FIG. 35 is a flow diagram of a timer interval manager process executed by the timer manager of FIG. 23.

DETAILED DESCRIPTION OF THE INVENTION

[0056] FIG. 2 is a block diagram of a storage server 30 according to an embodiment of the present invention. The storage server 30 includes a host processor 32, an ISA bridge 34, an ISA (industry standard architecture) bus 36, a network processor 38, a response bus 40, a Z0 ZBT (zero bus turnaround) SRAM (static random access memory) interface 42, a co-processor 44, a first external memory 46, a first DDR (double data rate) SRAM bus 48, a second external memory 50, and a DDR SRAM bus 52. The storage server 30 also includes other components (not shown), the details of which are not necessary for an understanding of the present invention.

[0057] The host processor 32 controls the configuration and general operation of the storage server 30. The ISA bridge 34 connects the host processor 32 to the co-processor 44 over the ISA bus 36.

[0058] The network processor 38 controls the routing of data frames into and out of the storage server 30. The network processor executes software that may also be referred to as "pico code".

The pico code controls the operation of the network processor 38. The network processor 38 is coupled to the co-processor 44 over the Z0 ZBT SRAM interface 42. The co-processor may also communicate with the network processor 38 via the response bus 40.

[0059] The co-processor 44 offloads some of the functions performed by the pico code on the network processor 38. The co-processor 44 may also be referred to as "the pico co-processor" (or more generally as "the processor" in context). The co-processor 44 may be implemented as a field programmable gate array (FPGA) device or programmable logic device (PLD) that has been configured to perform the desired functions.

[0060] The first external memory 46 and the second external memory 50 may each be implemented as 36 x 1M DDR SRAMs. Other size SRAMs and other memory technologies may also be used. The first external memory 46 and the second external memory 50 are coupled to the co-processor 44 via the first DDR SRAM bus 48 and the second DDR SRAM bus 52, respectively.

[0061] The Pico Co-Processor 44 is designed to offload some of the functions performed by the Pico code running inside the network processor. The Pico Co-Processor 44 is connected to the Network Processor 38 replacing the Z0 ZBT SRAM that may be customarily attached. Data and commands may be transferred to and from the Pico Co-Processor 44 using ZBT SRAM cycles. Some commands may issue a response to the Network Processor 38 using the support CAM (content addressable memory) response bus 40.

[0062] The Pico Co-Processor 44 may be configured and monitored by the host processor 32 over the ISA bus 36. It is also possible for the processors (not shown) in the network processor 38 to monitor the Pico Co-Processor 44 through the ISA bus 36.

[0063] The Pico Co-Processor 44 uses two DDR SRAM banks 46 and 50 for storing tables and other structures used by the various internal modules. According to one embodiment, the interfaces 48 and 52 between the Pico Co-Processor 44 and the two SRAM banks 46 and 50 operate at 133Mhz and are parity protected.

[0064] The following modules inside the Pico Co-Processor 44 provide offload functions for the Pico code of the network processor 38:

- Semaphore Manager: Manages locks on 32-bit values that are requested and released by Pico code as internal shared structures are locked and unlocked.

- Out of Order Processor: Assists Pico code by tracking out of order frames and returning pointers to frame data in the proper order.

- Timer Manager: Allows Pico code to create and delete general-purpose timers.

[0065] SEMAPHORE MANAGER

[0066] FIG. 3 is a block diagram of a semaphore manager 100 according to an embodiment of the present invention. The semaphore manager 100 is part of the processor 44 (see FIG. 2). The semaphore manager 100 includes an interface controller 102, a command FIFO 104, a hash key generator 106, an update engine 108, a hash bucket memory 110, a semaphore structure memory 112, and a free semaphore queue manager 114.

[0067] The interface controller 102 interfaces the semaphore manager circuitry to the network processor 38. The command FIFO 104 processes input data as it is received from the network processor 38. The input data from the network processor 38 is generally in the form of commands to the semaphore manager 100. The input data is generally 32 bits of command data and 7 bits of address data.

[0068] The hash key generator 106 generates a hash key from a semaphore value. According to the embodiment of FIG. 3, the semaphore value is 32 bits and the hash key is 10 bits.

[0069] The update engine 108 performs the main functions of semaphore processing, which are discussed in more detail below. The update engine communicates information back to the network processor 38 via the interface controller 102.

[0070] The hash bucket memory 110 stores numerous linked lists of active semaphore structures. The semaphore structure memory stores information regarding semaphores and the process threads on the network processor 38 that are waiting to lock the semaphores.

[0071] In general, the Semaphore Manager 100 allows the NP Pico code to lock and unlock 32-bit values over the Z0 interface 42. The results of the lock and unlock requests are passed back to the thread over the Z0 response bus 42.

[0072] The data structures used in the Semaphore Manager reside in internal FPGA block RAM, namely, the hash bucket memory 110 and the semaphore structure memory 112.

[0073] FIG. 4 is a representation of the hash bucket memory 110. The hash bucket memory 110 includes an array 120 of 11-bit bucket pointers that each serve as the head pointer of a linked list of active semaphore structures 122. The hash key generator 106 performs a hash function to convert the 32-bit semaphore value to a 10-bit hash address pointing to one of 1024 locations in the Hash Bucket Array 120.

[0074] FIG. 5 is a representation of a semaphore structure 124 stored in the semaphore structure memory 112. The semaphore structure 124 is a 128-bit entry in the semaphore memory 112. Generally, each of the semaphore structures 124 stores a locked semaphore value, the current thread that owns the lock, and a list of threads that are waiting to lock of the semaphore.

[0075] More specifically, the semaphore structure 124 contains the semaphore value, a pointer to the next semaphore in a linked list, the id of the owner thread running on the network processor 38, a count of requesting threads, and a list of requesting thread values. The pointer valid flag indicates if the pointer field contains a valid next semaphore pointer. The next semaphore pointer contains extra bits for future expansion. According to the embodiment of FIG. 5, each semaphore structure may store the current owner as well as 15 waiting threads. According to the embodiment of FIG. 5, the semaphore memory currently supports 512 entries. These numbers may be adjusted as desired in other embodiments.

[0076] SEMAPHORE MANAGER COMMANDS

[0077] Three commands supported by the Semaphore Manager 100 are "Semaphore Request", "Semaphore Release", and "Thread Exit".

[0078] A "Semaphore Request" command involves a 32-bit write to the request address associated with the requesting thread. The 32-bit value written is the value that is to be locked by the Semaphore Manager 100. If an error occurs in the processing of the lock request, an error response may be sent to the thread over the response bus 40. A "Semaphore Overflow" response is sent if the number of threads in line for the lock request exceeds the tracking ability of the semaphore structure 124 (15 threads according to the embodiment of FIG. 5). If no error or overflow occurs, the thread will receive a response when the thread has been issued the lock.

[0079] A "Semaphore Release" command involves a 32-bit write to the release address associated with the requesting thread. The 32-bit value written is the value that is to be released by the Semaphore Manager 100. If an error occurs when attempting to find the semaphore value, an error response will be sent to the thread over the response bus 40. If the release succeeds, a "Release Acknowledge" response will be sent to the thread over the response bus 40. If another thread was waiting in line for the semaphore, then a "Request Acknowledge" response will be sent to the next thread in line.

[0080] A thread requesting a semaphore lock will usually expect a response. A thread releasing a semaphore lock is not required to wait for a release response from the semaphore manager. The following table defines the response values received from the semaphore manager.

Value	Operation	Meaning
0x0	Request	Semaphore Request Acknowledge
0x1	Release	Semaphore Release Acknowledge
0x4	Release	Semaphore Release Fail
0x5	Request	Semaphore Structure Overflow, Retry Request
0x6	Release	Semaphore Release When Not Owner
0x7	Request	Semaphore Structure Allocation Failure, Retry Request

Table 1. Semaphore Manager Response Values

[0081] The third command support by the semaphore manager is "Thread Exit". This command is issued when the thread is about to exit. This command is used to detect a situation where a thread exits prematurely and still either owns a semaphore lock or is in line for a semaphore lock. If a "Thread Exit" command is issued and the thread still is active in one or more semaphores, the THREAD_ERR bit will be set and an interrupt can be asserted to either the Host Processor 32 or a processor in the network processor 38. The "Thread Exit" command need not have a response.

[0082] SEMAPHORE MANAGER OPERATION

[0083] FIG. 6 is a flow diagram of the "Semaphore Request" command. This process is started by a thread writing a 32-bit value to the Semaphore Request address space in the Z0 memory map.

[0084] In step 130a, the semaphore manager 100 receives the "Semaphore Request" command. In step 130b, the hash key generator 106 generates a 10-bit hash bucket address from the 32-bit semaphore value. In step 130c, the update engine 108 reads the value stored at the hash bucket address. In step 130d, the update engine 108 checks whether the pointer valid bit is set. If so, in step 130e, the update engine 108 updates the semaphore address (resulting from accessing the hash bucket memory 110) and performs a read from the semaphore structure memory 112. In step 130f, the update engine 108 determines whether the semaphore value matches that of the semaphore structure 124 read. If so, in step 130g, the update engine 108 checks the wait count for the semaphore structure 124. In step 130h, the update engine 108 compares the wait count to the maximum number of waiting threads allowed by the semaphore structure 124, which is 15 threads in the embodiment of FIG. 5. If the wait count has not reached its limit, in step 130i, the update engine 108 adds the requesting thread to the wait list in the semaphore structure 124 and increments the wait count. In step 130j, the update engine 108 writes the current semaphore data to the semaphore structure memory 112. In step 130k, the update engine 108 increments the active count for the requesting thread. The process is then complete for that command.

[0085] Step 130l occurs from step 130d if the pointer valid bit is not set (or from step 130s as described below). In step 130l, the update engine 108 allocates a new semaphore structure. In step 130m, the update engine 108 checks with the free semaphore queue manager 114 to see if the free queue is empty. If not, in step 130n, update engine 108 updates the new semaphore data. In step 130o, the update engine 108 writes the semaphore address to the previous data (either the bucket entry or the previous semaphore). In step 130p, the update engine 108 writes the current semaphore data to the semaphore structure memory 112. In step 130q, the update engine sends a request acknowledgement response to the requesting thread on the network processor 38 via the interface controller 102. The process then moves to step 130k as above.

[0086] Step 130r occurs if in step 130f the semaphore does not match. In step 130r, the update engine 108 looks at the next structure in the hash bucket memory 110 and checks the pointer

valid bit. In step 130s, if the pointer valid bit is set, the process then moves to step 130e as above. If the pointer valid bit is not set in step 130s, the process then moves to step 130l as above.

[0087] Step 130t occurs if in step 130m the free queue is empty. In step 130t, the update engine 108 increments the allocation error counter. In step 130u, the update engine 108 sends an allocation error response to the requesting thread on the network processor 38. The process is then complete for that command.

[0088] Step 130v occurs if in step 130h the wait count hits its maximum value. In step 130v, the update engine 108 increments the thread overflow counter. In step 130w, the update engine 108 sends a thread overflow response to the requesting thread on the network processor 38. The process is then complete for that command.

[0089] FIG. 7 is a flow diagram of the "Semaphore Release" command. This process is started by a thread writing a 32-bit value to the Semaphore Release address space in the Z0 memory map.

[0090] In step 132a, the semaphore manager 100 receives the "Semaphore Release" command. In step 132b, the hash key generator 106 generates a 10-bit hash bucket address from the 32-bit semaphore value. In step 132c, the update engine 108 reads the value stored at the hash bucket address. In step 132d, the update engine 108 checks whether the pointer valid bit is set. If so, in step 132e, the update engine 108 updates the semaphore address (resulting from accessing the hash bucket memory 110) and performs a read from the semaphore structure memory 112. In step 132f, the update engine 108 determines whether the semaphore value matches that of the semaphore structure 124 read. If so, in step 132g, the update engine 108 checks the current thread value for the semaphore structure 124. In step 132h, the update engine 108 verifies whether the current thread value corresponds to the thread sending the semaphore release command. If so, in step 132i, the update engine 108 checks the wait count for the semaphore structure 124. In step 132j, the update engine 108 compares the wait count to zero (that is, that there are no other threads in line for that semaphore). If so, in step 132k, the update engine 108 works with the free semaphore queue manager 114 to return the semaphore structure to the queue. In step 132l, the update engine 108 clears the address in the previous data (either the bucket entry or the previous semaphore). In step 132m, the update engine 108 decrements the

active count for the releasing thread. In step 132n, the update engine sends a release acknowledgement response to the releasing thread. The process is then complete for that command.

[0091] Step 132o occurs if in step 132j the wait count is not zero (that is, that other threads are in line for that semaphore). In step 132o, the update engine 108 shifts the wait list and assigns the new current thread value. The update engine 108 also decrements the wait count. In step 132p, the update engine 108 writes the updated semaphore data to the semaphore structure memory 112. The process then moves to step 132m as above.

[0092] Step 132q occurs if in step 132f the semaphore does not match. In step 132q, the update engine 108 looks at the next structure in the hash bucket memory 110 and checks the pointer valid bit. In step 132r, if the pointer valid bit is set, the process then moves to step 132e as above.

[0093] Step 132s occurs if the pointer valid bit is not set from step 132r or step 132d. In step 132s, the update engine 108 sets the release error flag. In step 132t, the update engine 108 sends a release error response to the requesting thread. The process is then complete for that command.

[0094] Step 132u occurs from step 132h if the current thread value does not correspond to the thread sending the semaphore release command. In step 132u, the update engine 108 sets the release error flag. In step 132v, the update engine 108 sends a release error response to the requesting thread. The process is then complete for that command.

[0095] FIG. 8 is a flow diagram of the "Thread Exit" command. This is a simple command that requires no memory access.

[0096] In step 134a, the semaphore manager 100 receives the "Thread Exit" command. In step 134b, the update engine 108 checks the active count for that thread. In step 134c, the update engine 108 compares the active count to zero (that is, that the thread is not in line for any semaphores). If so, the process is complete for that command. If not, in step 134d the update engine 108 sets the thread error flag, and the process is complete for that command. The storage server 30 may then initiate diagnostic procedures to clear the thread error flag.

[0097] SEMAPHORE MANAGER INITIALIZATION

[0098] The Semaphore Manager 100 may be initialized before the NP Pico code can begin requesting locks. If a thread requests a lock while the Semaphore Manager 100 is in initialization mode, the request may be lost and the SEM_LOST flag may be set.

[0099] Software begins the initialization process by setting the SEMM_RST bit in the Semaphore Manager Control register. Setting this bit holds the engines in reset. The SEMM_RST bit may then be cleared by software, starting the automatic initialization. The SEM_INIT bit in the status register allows software to detect when the initialization process has completed. Once the initialization process has completed, the NP Pico code may begin requesting and releasing semaphore locks.

[0100] SEMAPHORE MANAGER STATUS

[0101] The Semaphore Manager 100 has a number of counters and status bits that indicate the state of the internal semaphore logic. The following counters are kept in the Semaphore Manager 100 and may be read and cleared through the ISA bus interface:

- Thread Overflow Counter – This counter keeps track of the number of times the semaphore structure overflows. According to one embodiment, an overflow occurs when more than 16 threads have requested the same semaphore.

- Allocation Error Counter – This counter keeps track of the number of times a new semaphore lock is requested when the free queue of semaphore structures is empty.

[0102] In addition to the previously-described counters, a number of status flags are also available to software and can be read over the ISA bus interface:

- SEM_INIT – This flag is set when the Semaphore Manager is in initialization mode and should not be accessed over the Z0 interface.

- SEM_LOST – This flag is set when a semaphore value has been written over the Z0 interface and did not make it to the hash / search engine. This condition may occur if a semaphore release / request is attempted when the semaphore manager is in initialization mode. This condition may also occur if the input buffer to the hash stage overflows. The assertion of this bit can generate an interrupt to the host processor and / or the network processor.

- REL_FAIL – This flag is set when a thread has attempted to release a semaphore lock and the thread is not the current owner of the lock or the semaphore structure could not be found. The assertion of this bit can generate an interrupt to the host processor and / or the network processor.

- THREAD_ERR – This flag is set when a thread exit command is issued and the thread is still the owner or in line for a semaphore lock. The assertion of this bit can generate an interrupt to the host processor and / or the network processor.

[0103] ORDERING PROCESSOR

[0104] FIG. 9 is a block diagram of an ordering processor 200 according to an embodiment of the present invention. The ordering processor 200 is part of the processor 44 (see FIG. 2). The ordering processor 200 may also be referred to as an out-of-order processor. The ordering processor 200 includes an interface controller 202, an input FIFO 204, a command pre-processor 206, an update state machine 207, a memory controller 210, a read buffer 212, and a response interface 214.

[0105] The interface controller 202 interfaces the ordering processor circuitry to the network processor 38. The input FIFO 204 processes input data as it is received from the network processor 38. The input data from the network processor 38 is generally in the form of commands to the ordering processor 200. The input data is generally 32 bits of command data and 5 bits of address data.

[0106] The command pre-processor 206 pre-processes commands from the input FIFO 204. The update state machine 208 receives the pre-processed commands and controls the components of the ordering processor 200 to carry out the commands. The memory controller 210 interfaces the ordering processor 200 to the external memory 46 (see also FIG. 2). The read buffer 212 stores frame information to be provided to the network processor 38. The response interface 214 interfaces responsive data from the ordering processor 200 to the network processor 38 (for example, a response that a particular command has been carried out, or a response indicating an error).

[0107] The Ordering Processor 200 assists the network processor 38 in handling frames that are received out of order. When out of order frames are received, they are stored in local NP

memory until the frames that need to be sent first are received. The Ordering Processor 200 assists the NP 38 by tracking the received frames, storing associated data for the frames, and alerting the NP 38 when the frames are to be sent.

[0108] The data stored in the out of order processor 200 consists of frame structures stores in queues. The OP 200 supports a variable number of queues defined by the system processor at initialization time. As frames are received, the NP 38 identifies which queue number the frame is associated with. As frames are added to a queue, their associated frame structure may be added to a linked list that is sorted by frame number. At the head of this linked list is the queue head structure.

[0109] FIG. 10 is a diagram of one embodiment of the queue head structure 220. Each of the queues supported by the OP 200 has a head structure 220 associated with it. The number of supported queues is flexible and defined by setting the QUEUE_BASE and QUEUE_TOP configuration values. The address of the head pointer of a queue is determined by bit shifting the queue number and adding it to QUEUE_BASE. Each queue head structure 220 is 64 bits.

[0110] The queue head element 220 contains a pointer to the top frame element in the frame list, a pointer valid flag, a Transmit In Progress flag, the number of the expected frame number, and the number of the next frame currently stored in the frame list. The top frame pointer and valid flag are used to find the next element in the queue or to determine if the queue is currently empty. The 'T' bit indicates that the Network Processor 38 is currently transmitting a frame. When this bit is set, no new frames can be transmitted even if they match the expected frame number, ensuring that frames are not transmitted by the network processor 38 out of order. The expected frame value in the queue head structure 220 identifies the frame number that is to be transmitted by the NP 38. The Next frame number identifies the frame number of the next frame stored in the list.

[0111] As frames are received in the network processor 38, the Pico Code passes the received frame number, the address in which the frame is stored in the egress queue, the queue number that the frame is associated with, and some NP-specific data. The OP engine 200 checks the state of the queue and compares it with the information provided with the received frame. The OP 200 determines if the frame can be transmitted or if it should be stored in the OP queue to be sent at a later time.

[0112] The OP 200 takes the address in which the frame is stored in the NP 38 and generates the address for the frame structure in the OP memory. The generated value will be compared against the FRAME_TOP configuration value to ensure that it does not exceed the allocated space in OP memory.

[0113] By using this direct address generation method, the OP 200 can look for errors where a new frame is stored at a location before the frame that was previously stored there has been transmitted. This also eliminates the need for a free frame structure queue.

[0114] Once the address of the received frame buffer and the active state of the frame are checked, the OP 200 updates the frame structure associated with the NP memory location.

[0115] FIG. 11 is a diagram of a frame structure 222 according to an embodiment of the present invention. Each frame entry structure 222 contains a pointer to the next frame structure in the linked list, a pointer valid flag, the frame number of the next frame in the list, the frame number that this structure refers to, a frame active flag, and associated data for the frame. A 20-bit buffer address and 20-bit BCI data field are also stored.

[0116] The next frame value stores the frame number of the next frame in the list and is used if the next pointer valid flag is set. The frame active bit determines if the buffer location currently stores a frame waiting to be transmitted or if the buffer is currently empty. The buffer address and BCI data fields store NP-specific information that was passed along with the frame when the "Frame Received" command was issued. This data is passed back to the NP when the "Frame Poll" or "Frame Pop" commands are issued.

[0117] ORDERING PROCESSOR COMMANDS

[0118] The Ordering Processor 200 supports five separate commands that are used at various times during the operation of the OP queue. All of these commands are processed in the order that they are received, and each command has an associated NP response with it. Once a thread issues a command, it will wait for a response to be sent when the command has been completed. The five commands supported by the OP 200 are described below.

[0119] FIG. 12 is a diagram showing the format of the "Init Queue" command 224. The "Init Queue" command is used to initialize the state of a specific OP queue. This command is used to

reset the next frame field in the addressed queue as well as to determine if the queue needs to be cleaned up. If the queue is currently empty, the OP engine 200 will update the "Next Frame" field and send an "Init Success" response to the NP 38. The queue may then be used to store received frames. If they identified queue currently has frames in its linked list, an "Init Fail" response will be sent to the NP 38.

[0120] FIG. 13 is a diagram showing the format of the "Frame Pop" command 226 and related buffer data 228. The "Frame Pop" command is used to pull frame entries off of an OP queue. Each time the NP 38 issues this command, a response will be sent indicating if the queue is empty or if a frame has been popped off of the queue. If there are any frames currently on the identified queue, the first frame structure in the list is read and its frame information is moved to the read buffer for the calling thread. The active bit for the 'popped' frame is then cleared.

[0121] According to one embodiment, the read buffer data 228 is read in a shape of three words with the first 32-bit value being a NULL value. The shape of three-word read is used to increase the cycle time so the appropriate thread data can be routed to the read interface.

[0122] FIG. 14 is a diagram showing the format of the "Frame Received command 230. The "Frame Received" command is issued each time the NP 38 receives a frame that requires out of order processing. The NP 38 passes the queue number for the frame, the received frame number, the address of the NP storage location, and some associated data. The OP engine 200 accesses the identified queue and determines if the frame should be stored for later processing or if the NP can immediately send the frame. If the frame can be sent, a "Frame TX" response is sent, the NP will then transmit the frame, and the NP will follow up with a "Frame Transmit" command indicating that the frame has been transmitted. A "Frame Store" response is sent if the frame is to be stored for later transmission. If the OP 200 detects an error during its operation, it sends a "Frame Error" response and sets an error flag. No processing on the frame is generally performed if an error is detected.

[0123] FIG. 15 is a diagram showing the format of the "Frame Poll" command 232 and related buffer data 234. The "Frame Poll" command is used to check to see if the expected frame is currently stored in the linked list and ready for transmission. Each time the NP calls this command, a response is be sent indicating if the expected frame is ready to be transmitted. If the frame can be transmitted, its frame structure is read and its frame information is moved to the

read buffer for the calling thread. The active bit for the frame is cleared. If no frame is ready to be transmitted, a "No Frame" response will be sent.

[0124] The read buffer data 234 is read in a shape of three words with the first 32-bit value being a NULL value. The shape of three-word read is used to increase the cycle time so the appropriate thread data can be routed to the read interface.

[0125] FIG. 16 is a diagram showing the format of the "Frame Transmit" command 236. The "Frame Transmit" command is used to indicate that the transmission of the expected frame has been completed successfully. The NP 38 issues this command following a "Frame Received" command where it was told to transmit the frame, or following a successful "Frame Poll" command. When the OP engine 200 receives this command, it increments the expected frame value. The response to this command will be one of three values. The first two values indicate no error and encode whether or not the next frame to be transmitted is currently available in the buffer. If the next frame is available, the NP 38 will follow up with a "Frame Poll" command to get the frame information. If the frame value provided with the "Frame Transmit" command does not match the frame number that the OP 200 expected to be transmitted, a "TX Error" response is sent to the NP 38 and the corresponding error bit is set.

[0126] ORDERING PROCESSOR OPERATION

[0127] This section describes the internal operation that occurs within the OP engine 200 when each of the five supported commands are received. As the commands are received by the NP interface 202, they are added to the input command queue 204. The depth of the command queue 204 makes it impossible for the buffer to overflow if the NP software operates properly. If the buffer overflows for any reason and a command is lost, the CMD_LOST flag will be asserted. The assertion of this signal has the ability to generate an interrupt to the host processor or network processor 38. The address presented on the NP interface 202 is used to determine the command and owner thread. Bits 9-7 encode the command to be performed and bits 6-2 encode the thread that is performing the command. Bits 1-0 are always zero for all commands except the "Frame Received" command. For the "Frame Received" command, the value encode in these bits will be 0, 1 or 2 depending on which long word of the command is being written.

[0128] FIG. 17 is a flow diagram for the "Init Queue" command. As described earlier, this command checks the queue to determine if it is empty, sets the new expected frame value, and responds with the appropriate response. The "Init Fail" status bit may be set during the operation of this command. The setting of this bit has the ability to generate an interrupt to the host processor or the network processor.

[0129] In step 238a, the ordering processor 200 receives the "Init Queue" command. In step 238b, the update state machine 208 works with the memory controller 210 to read in the head structure for the identified queue. In step 238c, the update state machine 208 checks the pointer valid flag in the head structure. In step 238d, the update state machine 208 checks whether the pointer valid bit is set. If not, in step 238e, the update state machine 208 sets the new expected frame value and clears the TX bit. In step 238f, the update state machine 208 works with the memory controller 210 to write the head structure to the memory 46. In step 238g, the update state machine sends the initialization success response to the network processor 38 via the response interface 214.

[0130] If step 238d determines that the pointer valid bit is set, in step 238h the update state machine 208 sets the initialization failure flag. In step 238i, the update state machine 208 sends the initialization failure response to the network processor 38 via the response interface 214.

[0131] FIG. 18 is a flow diagram for the "Frame Pop" command. This command allows the NP 38 to remove entries from a Queue. Any frame entries 'popped' from the queue have their active bit cleared.

[0132] In step 240a, the ordering processor 200 receives the "Frame Pop" command. In step 240b, the update state machine 208 works with the memory controller 210 to read in the head structure for the identified queue. In step 240c, the update state machine 208 checks the pointer valid flag in the head structure. In step 240d, the update state machine 208 verifies whether the pointer valid bit is set. If so, in step 240e, the update state machine 208 works with the memory controller 210 to read in the current frame structure at the pointer address. In step 240f, the update state machine 208 writes the current frame data to the read buffer 212 for the thread. In step 240g, the update state machine 208 writes the current frame next pointer to the head structure next pointer. The update state machine 208 writes the current frame next frame number to the head structure next frame number. In step 240h, the update state machine 208 clears the

current frame active bit. In step 240i, the update state machine 208 works with the memory controller 210 to write the queue head structure to the memory 46. In step 240j, the update state machine 208 works with the memory controller 210 to write the current data structure to the memory 46. In step 240k, the update state machine sends the frame data ready response to the NP 38. The process is then complete for this command.

[0133] Step 240l results from step 240d when the pointer valid bit is not set. In step 240l, the update state machine 208 sends the empty queue response to the NP 38. The process is then complete for this command.

[0134] FIGS. 19-20 are a flow diagram for the "Frame Received" command. This command results in a frame being transmitted by the NP 38 or the insertion of the received frame in the identified queue. The "Frame Received" command asserts the FRAME_ERR bit if an error occurs during the processing of the command. This bit has the ability to assert an interrupt to the host processor or network processor 38.

[0135] The address of the frame structure for the received frame, FRAME_ADDR, is generated from the buffer address provided along with the "Frame Received" command, BUFF_ADDR. The equation below uses the buffer address mask, BUFF_MASK, the buffer address shift, BUFF_SHIFT and the frame structure base address, FRAME_BASE to generate the frame structure address. The resulting value is then compared with the frame structure top address, FRAME_TOP.

$$OFF\_ADDR = (BUFF\_ADDR \ \& \ BUFF\_MASK) >> BUFF\_SHIFT$$

$$FRAME\_ADDR = (OFF\_ADDR << 2) + FRAME\_BASE$$

[0136] In general, the flowchart of FIG. 19 determines if the frame is to be transmitted for stored, and the flowchart of FIG. 20 performs the insertion of the frame structure into identified queue.

[0137] In step 242a, the ordering processor 200 receives the "Frame Received" command. In step 242b, the update state machine 208 generates the received frame structure address from the buffer address. The update state machine 208 compares the address with the FRAME_TOP value. In step 242c, the update state machine 208 determines whether there is an address

overflow. If not, in step 242d, the update state machine 208 reads in the received frame structure. In step 242e, the update state machine 208 determines whether the frame is active. If not, in step 242f, the update state machine 208 updates the received frame data and sets the received frame active bit. In step 242g, the update state machine 208 reads in the head structure for the identified queue. In step 242h, the update state machine 208 checks the TX bit in the head structure. In step 242i, the update state machine 208 determines whether the TX bit is set. If not, in step 242j, the update state machine 208 compares the received frame number with the expected frame number. In step 242k, the update state machine 208 verifies whether the frame numbers match. If so, in step 242l, the update state machine 208 marks the TX bit in the head structure. In step 242m, the update state machine 208 works with the memory controller 210 to write the queue head structure to the memory 46. In step 242n, the update state machine 208 sends the frame transmit response to the NP 38. The process is then complete for that command.

[0138] Step 242o results when step 242c has identified an address overflow. In step 242o, the update state machine 208 sets the frame error flag. In step 242p, the update state machine sends the frame error response to the NP 38. The process is then complete for that command.

[0139] Step 242q results when step 242k has determined the frame numbers do not match. In step 242q, the update state machine 208 examines whether the received frame number is less than the expected frame number. If so, in step 242r the update state machine 208 sets the frame error flag. In step 242s, the update state machine 208 sends the frame error response to the NP 38. The process is then complete for that command.

[0140] If the TX bit is determined from step 242i to be set, or if the received frame number is not less than the expected frame number in step 242q, the process moves to FIG. 20.

[0141] In step 244a, the update state machine 208 checks the head next pointer valid flag. In step 244b, the update state machine 208 determines whether the pointer is valid. If so, in step 244c, the update state machine 208 compares the head next frame number with the received frame number. In step 244d, the update state machine 208 determines whether the frame numbers match. If not, in step 244e, the update state machine 208 determines whether the frame number is less. If so, in step 244f, the update state machine 208 updates the received frame next pointer to the head next pointer, and updates the received frame next number to the head next number. In step 244g, the update state machine 208 updates the head next pointer to point to the

received frame structure, and updates the head next frame number to the received frame number. In step 244h, the update state machine 208 works with the memory controller 210 to write the head data structure to the memory 46. In step 244i, the update state machine 208 works with the memory controller 210 to write the received data structure to the memory 46. In step 244j, the update state machine 208 sends the frame store response to the NP 38. The process is then complete for that command.

[0142] Step 244k results from step 244e (or step 244n, see below) when the frame number is not less. In step 244k, the update state machine 208 reads the next frame data in the chain, and sets the next frame data read as the current frame data. In step 244l, the update state machine 208 compares the current next frame number with the received frame number. In step 244m, the update state machine determines whether the frame numbers match. If not, in step 244n, the update state machine determines whether the frame number is less. If so, in step 244o, the update state machine 208 updates the received frame next pointer to the current next pointer, and updates the received frame next number to the current next number. In step 244p, the update state machine 208 updates the current next pointer to point to the received frame structure, and updates the current next frame number to the received frame number. In step 244q, the update state machine 208 works with the memory controller 210 to write the current data structure to the memory 46. The process then moves to step 244i.

[0143] Step 244r results from step 244b when the pointer is not valid. In step 244r, the update state machine 208 updates the head next pointer to point to the received frame structure, and updates the head next frame number to the received frame number. In step 244s, the update state machine 208 works with the memory controller 210 to write the queue head structure to the memory 46. In step 244t, the update state machine 208 works with the memory controller 210 to write the received data structure to the memory 46. In step 244u, the update state machine 208 sends the frame store response to the NP 38. The process is then complete for that command.

[0144] Step 244v results from step 244d when the frame numbers do not match. In step 244v, the update state machine 208 sets the frame error flag. In step 244w, the update state machine 208 sends the frame error response to the NP 38. The process is then complete for that command.

[0145] Step 244x results from step 244m when the frame numbers do not match. In step 244x, the update state machine 208 sets the frame error flag. In step 244y, the update state machine 208 sends the frame error response to the NP 38. The process is then complete for that command.

[0146] FIG. 21 is a flow diagram for the "Frame Poll" command. In this command, the OP engine 200 checks to see if the next frame can be sent. If it can be sent, the frame information is moved to the read buffer for the requesting thread.

[0147] In step 246a, the ordering processor 200 receives the frame poll command. In step 246b, the update engine 208 works with the memory controller 210 to read in the head structure for the identified queue from the memory 46. In step 246c, the update state machine 208 checks the pointer valid flag in the head structure. In step 246d, the update state machine 208 determines whether the pointer valid bit is set. If not, in step 246e, the update state machine 208 sends the no frame response to the NP 38. The process is then complete for that command.

[0148] Step 246f results from step 246d when the pointer valid bit is set. In step 246f, the update state machine 208 checks the transmit bit in the head structure. In step 246g, the update state machine 208 determines whether the transmit bit is set. If so, the process moves to step 246e. If not, in step 246h, the update state machine 208 compares the next frame number with the expected frame number. In step 246i, the update state machine 208 determines whether the frame numbers match. If not, the process moves to step 246e. If so, in step 246j, the update state machine 208 works with the memory controller 210 to read in the current frame structure at the pointer address from the memory 46. In step 246k, the update state machine 208 writes the current frame data to the read buffer 212 for the thread. In step 246l, the update state machine 208 writes the current frame next pointer to the head next pointer, and writes the current frame next frame number to the head next frame number. In step 246m, the update state machine 208 marks the transmit bit in the head structure. In step 246n, the update state machine 208 clears the current frame active bit. In step 246o, the update state machine 208 works with the memory controller 210 to write the queue head structure to the memory 46. In step 246p, the update state machine 208 works with the memory controller 210 to write the current data structure to the memory 46. In step 246q, the update state machine 208 sends the frame data ready response to the NP 38. The process is then complete for that command.

[0149] FIG. 22 is a flow diagram for the "Frame Transmit" command. This command indicates to the OP engine 200 that the expected frame has been transmitted and the expected frame value can be incremented. The OP engine 200 checks to make sure the transmitted frame value matches the expected frame value and that the transmit flag has been set. If the transmit flag has not been set or the transmitted frame number does not match the expected frame number, the transmit error flag will be set. This flag has the ability to generate and interrupt to the host processor or the network processor 38.

[0150] In step 248a, the ordering processor 200 receives the frame transmit command from the NP 38. In step 248b, the update state machine 208 works with the memory controller 210 to read in the head structure for the identified queue from the memory 46. In step 248c, the update state machine 208 checks the transmit flag in the head structure. In step 248d, the update state machine 208 determines whether the transmit bit has been set. If so, in step 248e, the update state machine 208 compares the transmitted frame number to the expected frame number. In step 248f, the update state machine determines whether the frame numbers match. If so, in step 248g, the update state machine 208 clears the transmit bit in the head structure. In step 248h, the update state machine increments the expected frame number. In step 248i, the update state machine 208 compares the next frame number with the expected frame number. In step 248j, the update state machine 208 determines whether the frame numbers match. If so, in step 248k the update state machine 208 sets the poll bit in response. In step 248l, the update state machine works with the memory controller 210 to write the head structure to the memory 46. In step 248m, the update state machine 208 sends the transmit acknowledgement response to the NP 38. The process is then complete for that command.

[0151] Step 248n results from step 248j when the frame numbers do not match. In step 248n, the update state machine clears the poll bit in response. The process then moves on to step 248l.

[0152] Step 248o results from step 248d when the transmit bit is not set, or from step 248f when the frame numbers do not match. In step 248o, the update state machine 208 sets the transmit error flag. In step 248p, the update state machine 208 sends the transmit error response to the NP 38.

[0153] ORDERING PROCESSOR INITIALIZATION

[0154] Software starts the initialization process by setting the INIT_RST bit in the control register. This bit holds the internal state machines in a reset state while the ISA configuration registers are set. Software then configures the QUEUE_BASE and QUEUE_TOP registers for the queue head memory. Each queue head entry consumes two address locations.

[0155] Software then configures the FRAME_BASE and FRAME_TOP registers for the frame structure memory. Each frame structure entry consumes four address locations. Along with these two variables, software also sets the BUFF_MASK and BUFF_SHIFT configurations used to generate the frame structure address from the buffer address provided by the network processor 38.

[0156] Once the configuration values have been set, software clears the INIT_RST bit and allows the OP initialization to complete. When initialization has completed, the OOOP_INIT flag in the OP status register clears.

[0157] ORDERING PROCESS STATUS

[0158] The ordering processor 200 has a number of status bits that indicate the state of the internal logic.

- OOOP_INIT – This flag may be set when the OP is in INIT mode and should not be accessed over the Z0 interface 42.

- CMD_LOST – This flag may be set when a command has been written over the Z0 interface 42 and did not make it into the command buffer 204. The assertion of this bit may generate an interrupt to the host processor and / or the network processor 38.

- INIT_FAIL – This flag may be set when an "Init Queue" command has been issued and the queue is not empty. The assertion of this bit may generate an interrupt to the host processor and / or the network processor 38.

- FRAME_ERR – This flag may be set when a "Frame Received" command has been issued and a processing error occurs. The assertion of this bit may generate an interrupt to the host processor and / or the network processor 38.

- TX_ERR – This flag may be set when a "Frame Transmit" command has been issued and a processing error occurs. The assertion of this bit may generate an interrupt to the host processor and / or the network processor 38.

[0159] TIMER MANAGER

[0160] FIG. 23 is a block diagram of a timer manager 300 according to an embodiment of the present invention. The timer manager 300 is part of the processor 44 (see FIG. 2). The timer manager 300 includes an interface controller 302, a command buffer and decoder 304, a main controller 306, a remover 308, a first timer interval manager 310, a second timer interval manager 312, an arbiter 314, a memory controller 316, an expired queue manager 318, and an expire FIFO 320.

[0161] The interface controller 302 interfaces the timer manager circuitry to the network processor 38. The command buffer and decoder 304 processes input data as it is received from the network processor 38. The input data from the network processor 38 is generally in the form of commands to the timer manager 300. The input data is generally 32 bits of command data and 8 bits of address data.

[0162] The main controller 306 controls the components of the timer manager 300 to implement the commands from the NP 38 and to manage the timers. The remover 308 removes expired timers.

[0163] The first timer interval manager 310 and second timer interval manager 312 each manage a different timer interval. According to the embodiment of FIG. 23, two intervals are supported. Other embodiments may implement more or less numbers of intervals as desired. The timer interval managers 310 and 312 manage doubly-linked lists of timers (as more fully described below). The timer interval managers 310 and 312 each include a memory for storing various timer structures, including a currently active timer, a previously active timer, and optionally a new timer to be inserted into the list. These concepts are more fully described below.

[0164] The arbiter 314 arbitrates access to the memory 50 (via the memory controller 316) between the remover 308, the first timer interval manager 310, the second timer interval manager 312, and the expired queue manager 318. It is undesired for two of such components to

simultaneously attempt to access the memory 50. The memory controller controls memory access between the elements of the timer manager 300 and the memory 50.

[0165] The expired queue manager 318 manages placing expired timers in the expire FIFO 320. The expire FIFO 320 stores expired timers until the expired timers have been accessed by the NP 38.

[0166] The Timer Manager (TIMM) 300 eliminates the software overhead associated with updating the timers within the NP 38 by implementing and managing them inside the co-processor 44. Software can start, stop and restart a timer within the TIMM 300 with a simple write to the Z0 interface 42. When a timer expires, it is placed on the expire queue 320 for software to come in to read.

[0167] These timers can be started at anytime, run in parallel, have different timeout values and can all expire at the same time. Each timer is implemented using a default time period of either 100ms or 1sec (corresponding to the first and second timer interval managers 310 and 312). When a timer is started, a loop count is included. The loop count indicates how many time periods will pass before the timer expires.

[0168] FIG. 24 is a diagram of a timer record 330. The timer records 330 are stored in the external memory 50 in a 128-bit timer record. Each timer record 330 contains up and down pointers that are used to create a doubly linked list of active timer records. The pointer valid flags are used to indicate the validity of the two pointers.

[0169] The original loop count value is stored in the timer record 330 to be used by the "Restart Timer" command. This allows the NP 38 to restart the timer without passing the original loop count associated with the timer.

[0170] The state bits, E & A, indicate if the timer is currently Idle, Active or Expired. The 16-bit timeout handler is a value that is passed to the NP software if the timer expires. The loop count value is preloaded when the timer is created and is decremented once each internal period. When this count reaches zero, the timer is expired and is moved to the expired queue. The restart bit indicates if a timer restart operation is to be preformed the next time the timer expires. Finally, the "Interval Time" value contains the 30-bit timestamp of when the timer interval will

be complete. This value is compared against the free running timer to determine if the loop count value should be decremented.

[0171] FIG. 25 is a diagram of a doubly-linked list 332 of timers. Each of the timer interval managers 310 and 312 manage a respective doubly-linked list of timers. The doubly linked list 332 includes a top timer, an end timer, and various active timers. The top timer is the timer currently being managed. The end timer is the timer that was previously being managed. The various active timers will become the top timer, respectively, as the timer interval manager moves through the doubly-linked list. Within each timer structure (see also FIG. 24), the up pointer U points to the "previous" timer in the doubly-linked list, and the down pointer D points to the "next" timer in the doubly-linked list.

[0172] When a timer is created, it is added to a doubly linked list in the appropriate interval queue. When a new timer is created, its expire time value is generated by adding the interval time to the current value of the free running time clock. It is then added to the end of the doubly linked list. Adding records in this fashion ensures that the entry at the top of the list will always be the first entry to expire. A copy of the top entry is stored in the timer manager and its expire time value is always compared against the current timestamp.

[0173] When the timer value indicates an expired event, the loop count is checked. If the loop count is not zero, then it will be decremented, and a new expire time value will be created. The timer manager 300 then loads the next entry in the circular list into its internal memory and starts comparing the timestamp of the next entry with the free running counter. If the loop count is zero, the timer will be removed from the linked list and added to the expired queue.

[0174] A more detailed explanation of the operation of the timer manager and its commands are described below.

[0175] TIMER MANAGER COMMANDS

[0176] The Timer Manager 300 supports a series of commands that are used to start and stop timers as well as retrieve and acknowledge timers that have been placed on the expired queue 320. The start timer, stop timer and expired acknowledge commands have an associated response. The address space for each command is four Z0 address locations wide, with bits 1-0 selecting the long word of the command. For commands that only require a single long word,

the high long words are ignored and can be left out of the command. Address bits 6-2 are used to identify the thread that is issuing the commands. This information is used to direct the response back to the calling thread.

[0177] FIG. 26 is a diagram of the structure of the "Start Timer" command 334 and its response. The "Start Timer" command is used to activate an idle timer and place it on the appropriate timer queue. The "Start Timer" command consists of a write of a shape of 3 to the Z0 interface. Bits 15-0 of the first long word written identify the timer that is to be started. Bit 16 of the first long word identify the timer internal as either 100ms or 1s. Bits 17-0 of the second long word written contain a loop count value. This value defines the number of 'INT' time intervals must pass before the timer expires. Bits 15-0 of the third long word contain a timeout handler that is passed back to software if the timer expires before it has been stopped.

[0178] If the "Start Timer" command is successful, a "Start Ack" response is issued to the calling thread. If the timer is already running or if the timer is out of the range of configured timers, then a "Start Error" response is issued to the calling thread, and the "TIMER_ERR" error bit is set.

[0179] FIG. 27 is a diagram of the structure of the "Stop Timer" command 336 and its response. The "Stop Timer" command is used to stop a timer that is currently running. The stopped timer is removed from the timer queue and is marked as idle. The "Stop Timer" command consists of a write of a shape of 1 to the Z0 interface. Bits 15-0 of the long word written identify the timer that is to be stopped. If the "Stop Timer" command is successful, a "Stop Ack" response is issued to the calling thread. If the timer is not running or if the timer is out of the range of configured timers, then a "Stop Error" response is issued to the calling thread, and the "TIMER_ERR" error bit is set. If the timer is currently in the "Expired" state, then a "Stop Expired" response is sent.

[0180] FIG. 28 is a diagram of the structure of the "Restart Timer" command 338 and its response. The "Restart Timer" command is used to restart a timer that is currently running. If the timer has not expired, the loop count is reset to its initial value. The timer entry may not be moved from its current location in the active timer list. Because the timer entry is not moved in the list, the first decrement of the loop count may come earlier than the time interval. The "Restart Timer" command consists of a write of a shape of 1 to the Z0 interface. Bits 15-0 of the

long word written identify the timer that is to be restarted. If the "Restart Timer" command is successful, then a "Restart Ack" response may be sent to the NP 38. If the timer is not currently running, then a "Restart Error" response may be sent and the "Timer Error" flag may be set. If the timer is currently in the "Expired" state, then a "Restart Expired" response may be sent to the NP 38 indicating that the timer cannot be restarted.

[0181] FIG. 29 is a diagram of the structure of the "Read Expired" command 340. The "Read Expired" command allows the NP 38 to retrieve one or more timers that have expired. A read expired command consists of a 64-bit read, shape = 2, from the Z0 interface. Each read may return the next entry in the expired queue. The first 32-bit value read will always be NULL. If an entry read from the expired queue is valid, bit 16 in the second 32-bit value will be set. Bits 15-0 of the second 32-bit value contain the timer ID that has expired. If no more entries are available on the expired queue, bit 16 of the second 32-bit value will be clear. Reading an entry from the expired queue does not change the state of the expired timer. In order to return the expired timer back to "Idle", a "Clear Expired" command should be issued.

[0182] FIG. 30 is a diagram of the structure of the "Clear Expired" command 342 and its response. The "Clear Expired" command is issued following a read from the expired queue 320. When a timer expires, it is added to the expired queue 320 and set in the "Expired" state. It stays in this state until the NP 38 issues an "Clear Expired" command, indicating that the expiration has been acknowledged and that the appropriate actions have been taken. The "Clear Expired" command consists of a write in the shape of '1' to the "Clear Expired" command address. Bits 15-0 of the written data contain the timer ID of the timer that will be returned to the "Idle" state. If the identified timer is not in the "Expired" state, a "Clear Error" response will be sent to the NP 38, and the "TIMER_ERR" error bit will be set. Otherwise a "Clear Ack" response will be sent.

[0183] TIMER MANAGER OPERATION

[0184] This section describes the internal operation that occurs within the TIMM engine 300 when each of the supported commands is received. As the commands are received by the NP interface of the Pico Co-Processor 44 they are added to the input command queue. The depth of the command queue makes it impossible for the buffer to overflow if the NP software operates properly. If the buffer overflows for any reason and a command is lost, the CMD_LOST flag

will be asserted. The assertion of this signal has the ability to generate an interrupt to the host processor or network processor 38.

[0185] The address presented on the NP interface is used to determine the command and owner thread. Bits 8-7 encode the command to be performed, and bits 6-2 encode the thread that is performing the command. Bits 1-0 is always zero for all commands except the "Start Timer" command. For the "Start Timer" command, the value encode in these bits will be 0 or 1 depending on which long word of the command is being written.

[0186] FIG. 31 is a flow diagram of the "Start Timer" command. As described earlier, this command starts a new timer and adds it to the active timer list. An error response is sent if the identified timer is not currently in the "Idle" state or if the indicated timer is out of range. The main controller 306 generally performs the "Start Timer" command.

[0187] In step 344a, the timer manager 300 receives the start timer command from the NP 38. In step 344b, the timer manager 300 generates a timer pointer address from the timer ID, and checks if the generated address is greater than MEM_TOP_ADR. In step 344c, the timer manager 300 determines whether the address is greater. If not, in step 344d, the timer manager 300 works with the arbiter 314 and memory controller 316 to read in the timer record ("NEW") from the external memory 50 and checks the timer state. In step 344e, the timer manager 300 determines whether the timer is active. If so, in step 344f, the timer manager 300 updates the timer record with the provided data, generates the expire time value, and sets the timer as active. In step 344g, the timer manager 300 works with the arbiter 314 and memory controller 316 to read in the timer at the top of the active list ("TOP"). In step 344h, the timer manager 300 works with the arbiter 314 and memory controller 316 to read in the timer at the end of the active list ("END"). In step 344i, the timer manager 300 sets the UP pointer in TOP to point to the NEW record, and sets the DOWN pointer in END to point to the NEW record. In step 344j, the timer manager 300 sets the UP pointer in NEW to point to END, and sets the DOWN pointer in NEW to point to TOP. In step 344k, the timer manager 300 works with the arbiter 314 and the memory controller 316 to write the NEW, END and TOP data back to the memory 50. In step 344l, the timer manager 300 sends a start acknowledgement response to the NP 38. The process is then complete for that command.

Step 344m results from step 344c when the address is greater, or from step 344e when the timer is not active. In step 344m, the timer manager 300 sets the timer error flag. In step 344n, the timer manager 300 sends the start error response to the NP 38. The process is then complete for that command.

[0188] FIG. 32 is a flow diagram for the "Stop Timer" command. As described earlier, this command stops a currently running timer and marks it as Idle. The timer is removed from the active list. An error response is sent if the identified timer is not currently in the "Active" state or if the indicated timer is out of range. The main controller 306 generally performs the "Stop Timer" command. The remover 308 may be used to delete an active timer.

[0189] In step 346a, the timer manager 300 receives the stop timer command from the NP 38. In step 346b, the timer manager 300 generates the timer pointer address from the timer ID, and checks if the generated address is greater than MEM_TOP_ADR. In step 346c, the timer manager 300 determines whether the address is greater. If not, in step 346d, the timer manager 300 works with the arbiter 314 and the memory controller 316 to read in the timer record ("STP" record). The timer manager 300 also checks the timer state. In step 346e, the timer manager 300 determines if the timer has expired. If not, in step 346f, the timer manager 300 determines if the timer is active. If so, in step 346g, the timer manager 300 works with the arbiter 314 and the memory controller 316 to read in the timer pointed to by the UP pointer in STP ("UP" record). In step 346h, the timer manager 300 works with the arbiter 314 and the memory controller 316 to read in the timer pointed to by the DOWN pointer in STP ("DWN" record). In step 346i, the timer manager 300 sets the DOWN pointer in the UP record to point to the DOWN record, and sets the UP pointer in the DOWN record to point to the UP record. In step 346j, the timer manager clears the STP record data and marks it as inactive. In step 346k, the timer manager 300 works with the arbiter 314 and the memory controller 316 to write the STP, UP and DWN records back to the memory 50. In step 346l, the timer manager 300 sends the stop acknowledgement response to the NP 38. The process is then complete for that command.

[0190] Step 346m results from step 346c when the address is greater. In step 346m, the timer manager 300 sets the timer error flag. In step 346n, the timer manager 300 sends the stop error response to the NP 38. The process is then complete for that command.

[0191] Step 346o results from step 346e when the timer is expired. In step 346o, the timer manager 300 sends the stop expired response to the NP 38. The process is then complete for that command.

[0192] Step 346p results from step 346f when the timer is not active. In step 346p, the timer manager 300 sets the timer error flag. In step 346q, the timer manager 300 sends the stop error response to the NP 38. The process is then complete for that command.

[0193] FIG. 33 is a flow diagram of the "Restart Timer" command. As described earlier, this command takes a currently running timer and resets its loop count value. Because the loop count value is updated and the timer is left in its current location in the active timer list, there may be some slop in the total timer time. For example, if the timer being restarted is near the top of the active list, the first loop count decrement will occur right away. This will result in the timer expiring one interval earlier than expected. An error response is sent if the identified timer is not currently in the "Active" state or if the indicated timer is out of range. The main controller 306 generally performs the "Restart Timer" command.

[0194] In step 348a, the timer manager 300 receives the restart timer command from the NP 38. In step 348b, the timer manager 300 generates the timer pointer address from the timer ID, and checks if the generated address is greater than MEM_TOP_ADR. In step 348c, the timer manager 300 determines whether the address is greater. If not, in step 348d, the timer manager 300 works with the arbiter 314 and the memory controller 316 to read in the timer record ("RST" record). The timer manager 300 also checks the timer state. In step 348e, the timer manager 300 determines if the timer has expired. If not, in step 348f, the timer manager 300 determines if the timer is active. If so, in step 346g, the timer manager 300 updates the loop count to the original loop count value. In step 346h, the timer manager 300 works with the arbiter 314 and the memory controller 316 to write the RST data back to the memory 50. In step 348i, the timer manager 300 sends the restart acknowledgement response to the NP 38. The process is then complete for that command.

[0195] Step 348j results from step 348c when the address is greater. In step 348j, the timer manager 300 sets the timer error flag. In step 348k, the timer manager 300 sends the restart error response to the NP 38. The process is then complete for that command.

[0196] Step 348l results from step 348e when the timer is expired. In step 348l, the timer manager 300 sends the restart expired response to the NP 38. The process is then complete for that command.

[0197] Step 348m results from step 348f when the timer is not active. In step 348m, the timer manager 300 sets the timer error flag. In step 348n, the timer manager 300 sends the restart error response to the NP 38. The process is then complete for that command.

[0198] FIG. 34 is a flow diagram for the "Clear Expired" command. As described earlier, this command changes the state of an "Expired" timer to "Idle". This command is issued by the NP 38 after it has performed the appropriate actions following a timer expire event. An error response is sent if the identified timer is not currently in the "Active" state or if the indicated timer is out of range. The main controller 306 generally performs the "Clear Expired" command.

[0199] In step 350a, the timer manager 300 receives the clear expired command from the NP 38. In step 350b, the timer manager 300 generates the timer pointer address from the timer ID, and checks if the generated address is greater than MEM_TOP_ADR. In step 350c, the timer manager 300 determines whether the address is greater. If not, in step 350d, the timer manager 300 works with the arbiter 314 and the memory controller 316 to read in the timer record ("EXP" record). The timer manager 300 also checks the timer state. In step 350e, the timer manager 300 generates the timer pointer address from the timer ID. The timer manager 300 works with the arbiter 314 and the memory controller 316 to read in the timer record ("EXP" record). The timer manager 300 checks the timer state. In step 350f, the timer manager 300 determines if the timer is expired. If so, in step 350g, the timer manager 300 sets the EXP state to Idle. In step 350h, the timer manager 300 works with the arbiter 314 and the memory controller 316 to write the EXP data back to the memory 50. In step 350i, the timer manager 300 sends the clear acknowledge response to the NP 38. The process is then complete for that command.

[0200] Step 350j results from step 350c when the address is greater or from step 350f when the timer is not expired. In step 350j, the timer manager 300 sets the timer error flag. In step 350k, the timer manager sends the clear error response to the NP 38. The process is then complete for that command.

[0201] FIG. 35 is a flow diagram generally showing the operation of the timer interval managers 310 and 312. In addition to the command processing, the Timer Manager also has a Timer Engine that performs periodic updates the entries in the active timer list. This timer engine compares the expire time value of the entry at the top of the list with the free running time counter. When the expire time value is greater than or equal to the free running counter, the Timer Engine then processes the timer entry at the top of the list. The next entry in the list is then marked at the top entry and the comparison continues. These timer engine functions may generally be performed by the timer interval managers 310 and 312, which may each also be referred to as the timer engine.

[0202] In order to maximize the performance of the Timer Engine, the entry at the top of the list may be shadowed in internal memory. As soon as the Timer Engine processes the timer, a new entry may be read into the shadow memory. If the timer at the top of the list is acted on by a "Stop Timer" command, the next entry in the chain may be read in.

[0203] While the Timer Engine is active, any commands may be stalled until the updates are completed. Commands may then be processed while the Timer Engine is waiting for the top entry in the active list to expire.

[0204] An error indication that may occur during the Timer Engine operation is if an entry is to be added to the expired queue and the queue is currently full. In this case, the EXP_OFLOW flag may be sent that can cause an interrupt to be raised to the host processor or network processor 38. According to one embodiment, the expired FIFO can hold 1024 entries.

[0205] In step 352a, the timer engine comes out of initialization. In step 352b, the timer engine works with the arbiter 314 and the memory controller 316 to read in the TOP record. In step 352c, the timer engine compares the TOP entry expire time with the free running timer. In step 352d, the timer engine determines if the timer is expired. If not, the timer engine returns to step 352c. If so, in step 352e, the timer engine checks the restart flag. In step 352f, the timer engine determines if the flag is set. If not, in step 352g, the timer engine checks the loop count. In step 352h, the timer engine determines whether the loop count is zero. If so, in step 352i, the timer engine works with the arbiter 314 and the memory controller to read in the timer pointed to by the UP pointer in TOP ("UP"). In step 352j, the timer engine works with the arbiter 314 and the memory controller to read in the timer pointed to by the DOWN pointer in TOP ("DWN").

In step 352k, the timer engine sets the DOWN pointer in UP to point to the DWN record, and sets the UP pointer in DWN to point to the UP record. In step 352l, the timer engine marks TOP as expired, and works with the expire queue manager 318 to write the timeout handler to the expire queue 320. In step 352m, the timer engine determines whether the expired queue 320 is full. If not, in step 352n, the timer engine works with the arbiter 314 and the memory controller 316 to write the TOP, UP and DWN data back to the memory 50. In step 352o, the timer engine updates the TOP pointer to the DWN record and moves back to step 352c to continue the process.

[0206] Step 352p results from step 352f when the flag is set. In step 352p, the timer engine sets the loop count to the original loop count and clears the restart flag. In step 352q, the timer engine generates the new expire time value. In step 352r, the timer engine works with the arbiter 314 and the memory controller 316 to write the timer record to memory. The timer engine also updates the TOP pointer to the next entry in the chain. The process then moves back to step 352b to continue the process.

[0207] Step 352s results from step 352h when the loop count is not zero. In step 352s, the timer engine decrements the loop counter value. The process then moves back to step 352q to continue the process.

[0208] Step 352t results from step 352m when the expired queue is full. In step 352t, the timer engine sets the EXP_OFLOW flag. The process then moves back to step 352n to continue the process.

[0209] TIMER MANAGER INITIALIZATION

[0210] Software starts the initialization process by setting the INIT_RST bit in the control register. This bit holds the internal state machines in a reset state while the ISA configuration registers are set.

[0211] When the Timer Manager 300 is held in the "Init" state, software will program two variables via ISA, MEM_BASE_ADR and MEM_TOP_ADR, which set the range of memory space on the external memory allocated for the Timer Manager 300. These two parameters determine the number of timers that the module should support. These addresses are the absolute addresses of external memory 1. The base address is the first queue entry location and may

always be an even address. The top address is the location of the last queue entry and may always be an odd address. Each timer takes four memory words, so the difference between the two addresses divided by four gives the number of timers supported. In other embodiments, these parameters may differ.

[0212] Once the configuration values have been set, software will clear the INIT_RST bit and allow the Timer Manager initialization to complete. When initialization has completed, the TIMM_INIT flag in the Timer Manager status register will clear.

[0213] TIMER MANAGER STATUS

[0214] The Timer Manager has a number of status bits that indicate the state of the internal logic.

- TIMM_INIT – This flag may be set when the Timer Manager is in INIT mode and should not be accessed over the Z0 interface 42.

- CMD_LOST – This flag may be set when a command has been written over the Z0 interface 42 and did not make it into the command buffer 304. The assertion of this bit can generate an interrupt to the host processor and / or the network processor 38.

- TIMER_ERR – This flag may be set when a command has been issued and failed due to an unexpected state in the timer record. If a "Start Timer" command is issued and the timer is not "Idle", if a "Stop Timer" command is issued and the timer is not "Active" or "Expired", or if a "Clear Expired" command is issued and the state is not "Expired", this bit may be set. The assertion of this bit may generate an interrupt to the host processor and / or the network processor 38.

- EXP_OFLOW - This flag may be set when a failed attempt to add an entry to the expired queue has occurred. This may happen if the expire queue is full and another timer expires. The assertion of this bit may generate an interrupt to the host processor and / or the network processor 38.

[0215] Although the above description has focused on specific embodiments of the present invention, various modifications and their equivalents are considered to be within the scope of the present invention, which is defined by the following claims.